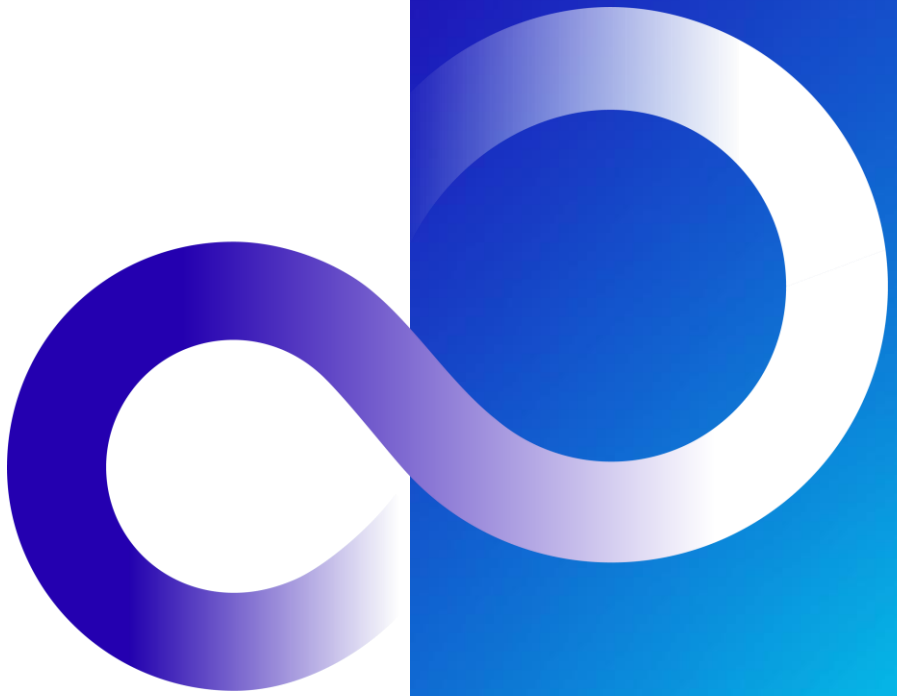
PostgreSQL
レプリケーション徹底解説

黒田 隼人
富士通株式会社
PostgreSQL Conference Japan 2025



自己紹介

FUJITSU

- 黒田 隼人
- 富士通株式会社 データマネジメント事業部
- PostgreSQL Contributor (2025/01-)
- 主な貢献：レプリケーション周りの改善
  - 大規模トランザクションの並列適用
  - pg_upgrade対応
  - その他バグ修正多数
- 講演・著作
  - **Logical Replication – handling of large transactions**
    (PostgreSQL Conference 2023)
  - **Online Upgrade of replication clusters without downtime**
    (PostgreSQL Development Conference 2024)
  - **Software Design誌 2025年2月号 寄稿**

本講演のゴール

FUJITSU

- PostgreSQLにおける2種類のレプリケーションを理解する
- 基本的なレプリケーション構成を構築できるようになる
- システムに最適なレプリケーション方法を選択できるようになる

FUJITSU

レプリケーションの基礎

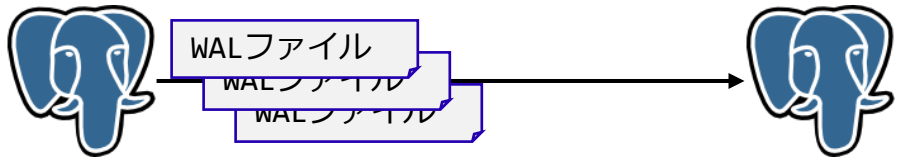
2種のレプリケーション
WAL (Write Ahead Log)
レプリケーションスロット

2種類のレプリケーション

ストリーミングレプリケーション

- プライマリはWALそのものを転送
- スタンバイはWALに記載された操作を実行

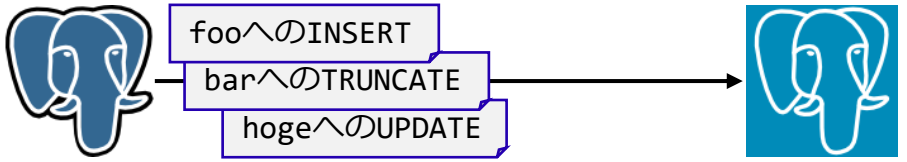
- プライマリとスタンバイは同一構成
- 高可用性・リアルタイム性重視



論理レプリケーション

- パブリッシャーはWALをデコードし、必要な変更情報を抽出して転送
- サブスクライバーは変更情報を元に反映

- テーブル単位で柔軟に制御可能
- データ移行・異種環境連携向け



レプリケーションの比較

比較項目	ストリーミングレプリケーション	論理レプリケーション
上流/下流インスタンスの呼び方	プライマリ/スタンバイ	パブリッシャ/サブスクライバ
ノード間で流れるデータ	**WALそのもの**	**WALから抽出された変更情報**
複製する対象	インスタンス全体	テーブル・スキーマ単位で指定 最大でもDB全体
上流インスタンスの数	1つだけ	複数設定が可能 サブスクリプションを複数持つことも可能
インスタンスの環境差異	同一OS・メジャーバージョンのみ	異なるOS・メジャーバージョン間でも可
制限事項	スタンバイでデータ更新不可	一部データ(DDL・シーケンス etc)は複製不可

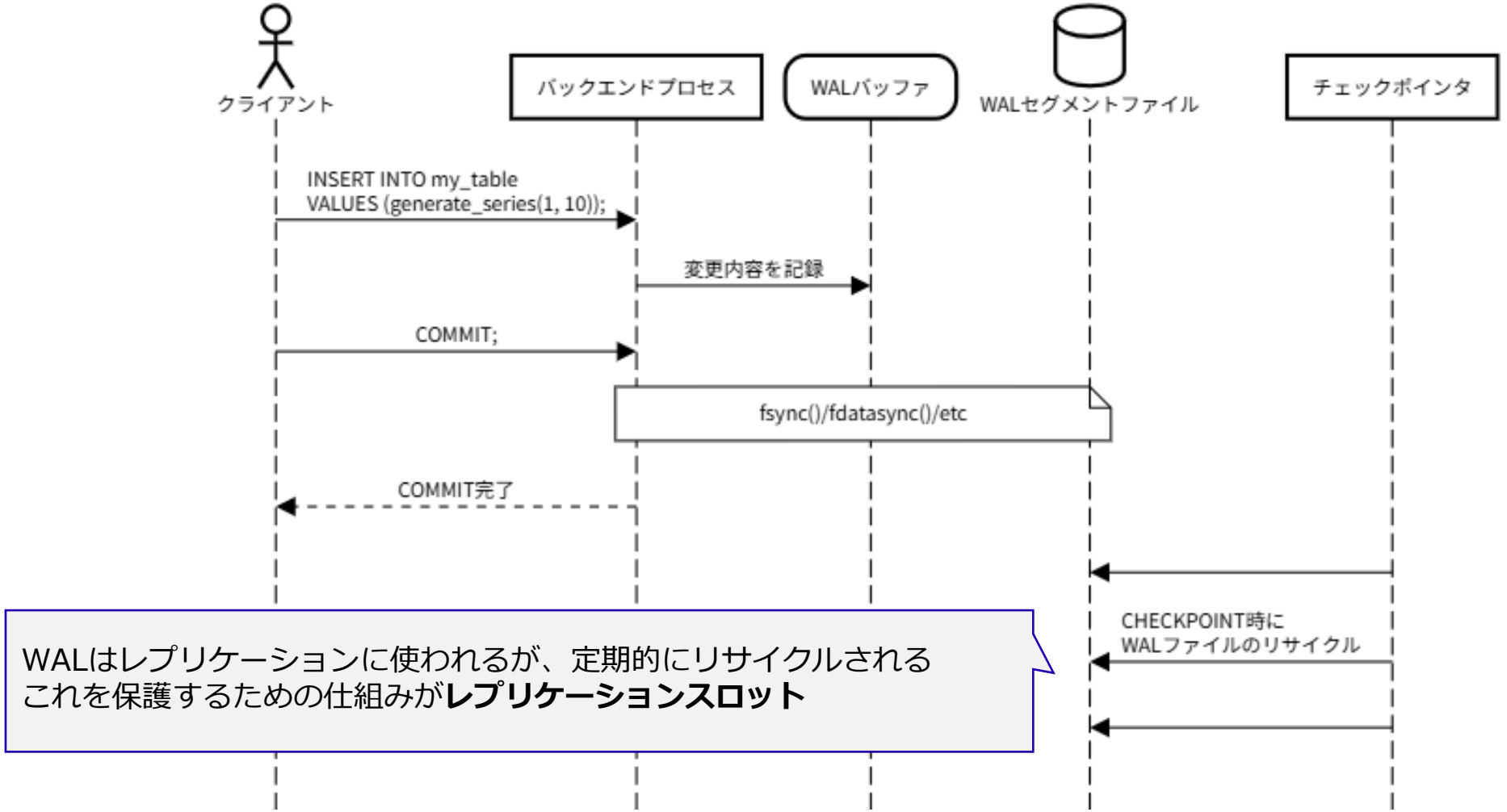
プライマリ/スタンバイで完全に同一
HA構成による可用性向上

異種環境でも動作
アップグレード時のデータ移行

WAL (Write Ahead Log) の基礎

FUJITSU

○データ変更を即時にテーブルファイルへ反映せず、先にログへ書く仕組み



WALはレプリケーションに使われるが、定期的にリサイクルされる
これを保護するための仕組みが**レプリケーションスロット**

実際のWALの中身

FUJITSU

```
postgres=*# INSERT INTO foo VALUES (1);
INSERT 0 1
postgres=*# SELECT txid_current();
 txid_current
--------------
          770
(1 row)

postgres=*# COMMIT;
COMMIT
```

WALの中身を表示するコマンド

ファイル名

```
$ pg_waldump primary/pg_wal/000000010000000000000001 -x 770
rmgr: Heap        len (rec/tot):     59/    59, tx:        770, lsn: 0/017C6560,
prev 0/017C6528, desc: INSERT off: 3, flags: 0x00, blkref #0: rel 1663/5/16384 blk 0

rmgr: Transaction len (rec/tot):     34/    34, tx:        770, lsn: 0/017C65A0, prev
0/017C6560, desc: COMMIT …
```

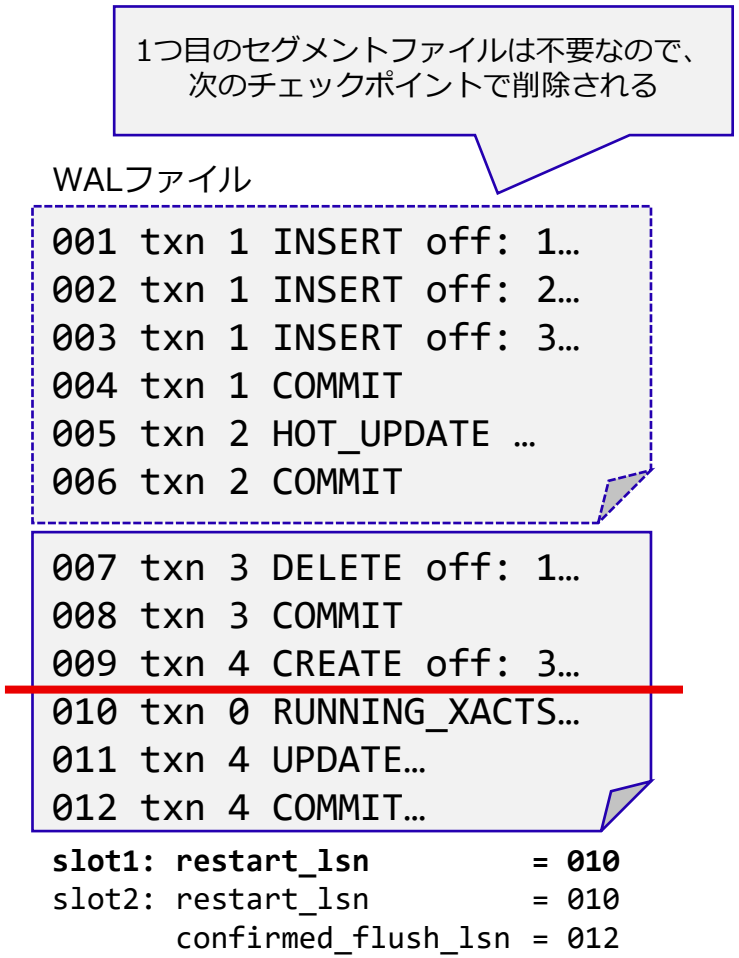
ページ内での挿入位置

レプリケーションスロット

FUJITSU

- どこまでのWALを読み取って出力したのかを記録
  - チェックポイント時に未送信のWALファイルが削除されるのを防ぐ
- 2種類のレプリケーションスロットが存在
  - 物理レプリケーションスロット：ストリーミングレプリケーションに使用
  - 論理レプリケーションスロット：論理レプリケーションに使用
- レプリケーションによってWALが転送された後に更新される
- すべてのスロットが不要と判断したWALは削除される
  - スロットが保護しているWALレコードは以下のSQLでわかる

```
postgres=# SELECT slot_name, pg_walfile_name(restart_lsn)
FROM pg_replication_slots;
   slot_name    |     pg_walfile_name
----------------+--------------------------
 physical_slot | 000000010000000000000004
(1 row)
```

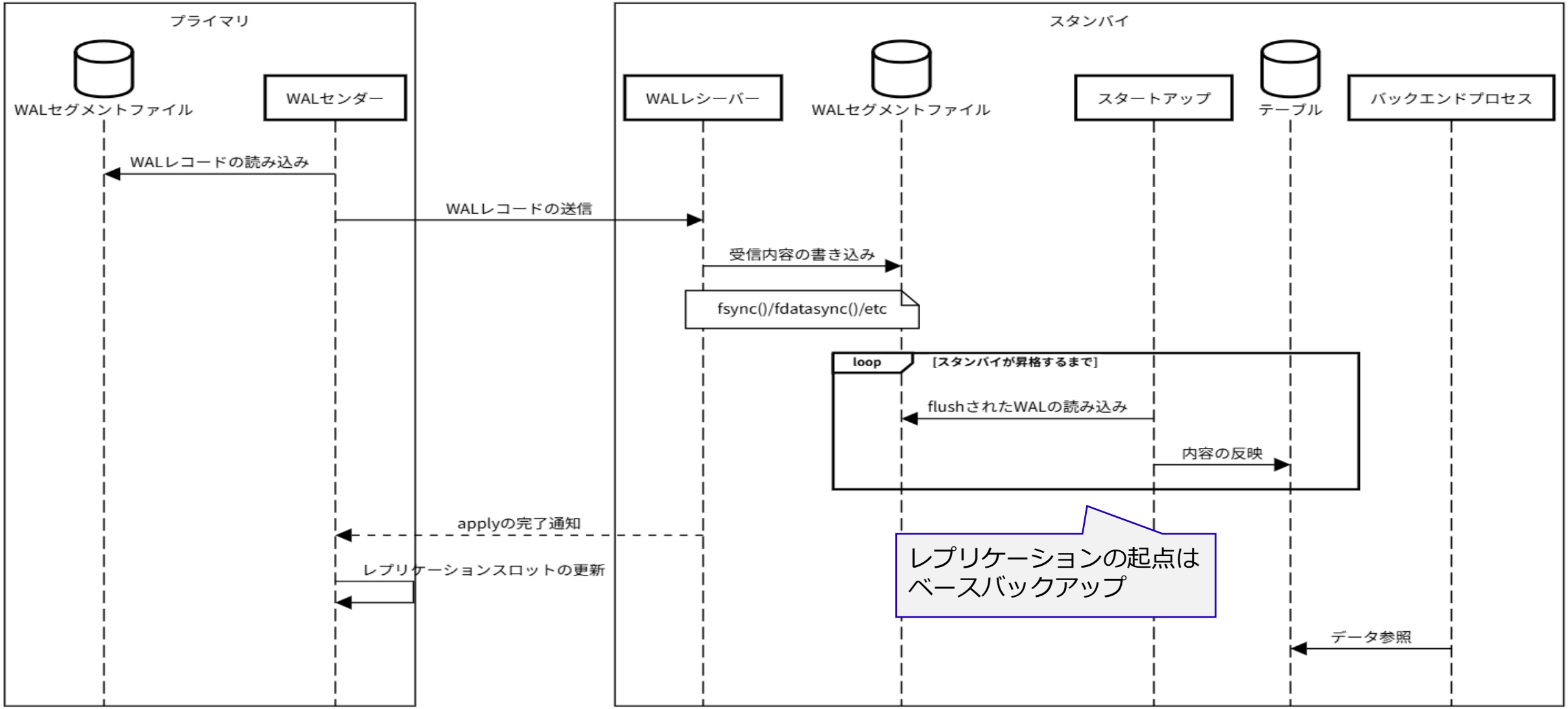


FUJITSU

ストリーミングレプリケーション

構成・動作
設定パラメータ
pg_basebackupによる環境の構築
利点と注意点

ストリーミングレプリケーションの構成と動作



主なパラメータ：ストリーミングレプリケーション編

FUJITSU

パラメータ名	解説	設定するインスタンス
wal_level	WAL出力の粒度 replicaまたはlogicalを指定 (minimalだとレプリケーションができない)	プライマリ
max_wal_senders	WALセンダープロセスの最大数 スタンバイの数以上に設定	プライマリ
max_replication_slots	レプリケーションスロットの最大数	プライマリ
synchronous_standby_names	同期レプリケーションを設定するスタンバイ名	プライマリ
synchronized_standby_slots	スロットを同期するなら追加で設定	プライマリ
primary_conninfo	プライマリサーバへの接続文字列	スタンバイ
hot_standby	スタンバイでの参照を受け付けるかどうか？	スタンバイ
primary_slot_name	このスタンバイとのレプリケーションで使用するスロット名	スタンバイ
hot_standby_feedback	プライマリにフィードバックを送るか？ スロットを同期するなら追加で設定	スタンバイ
sync_replication_slots	スロット同期を自動的に行うか？	スタンバイ

ストリーミングレプリケーション環境の構築

FUJITSU

```
$ pg_basebackup -d "user=postgres" -D standby -R
$ echo "port=5433" >> standby/postgresql.conf
$ pg_ctl start -D standby -l log.standby
waiting for server to start.... done
server started
$ ps auxf | grep postgres
…|       ¥_ /usr/local/pgsql/bin/postgres -D primary
…
…|       |   ¥_ postgres: walsender postgres [local] streaming 0/06000168

…|       ¥_ /usr/local/pgsql/bin/postgres -D standby
…
…|           ¥_ postgres: startup recovering 000000010000000000000006
…|           ¥_ postgres: walreceiver streaming 0/06000168
```

-R
--write-recovery-conf

バックアップ取得に使用した接続情報を
primary_conninfoに残す
さらにstandby.signalファイルを作成

※ プライマリ・スタンバイが同一サーバに存在する環境で実施

特徴と利点

- シンプルな構成
  - 単一のプライマリと複数のスタンバイから構成
  - 同一バージョン間で完結し、セットアップが容易
- 参照の負荷分散が可能
  - ホットスタンバイとして設定すると、接続を受け付けデータ参照が可能
- 処理が高速
  - 生成されたWALをそのまま転送する
  - 反映時もトランザクションの開始/終了を必要としない
- フェイルオーバーが簡単
  - SQL関数pg_promote()またはコマンドpg_ctl promoteを実行
  - 昇格自体は非常に高速
  - 昇格のタイミングは運用者・外部ツールで決定する必要がある

```
postgres=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 t
(1 row)

Time: 0.602 ms
postgres=# SELECT pg_promote();
 pg_promote
------------
 t
(1 row)

Time: 174.021 ms
postgres=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 f
(1 row)

Time: 0.248 ms
```

リカバリ中かどうか？
= ホットスタンバイかどうか？

制約と注意点

- インスタンス全体の複製のみ
  - 特定のデータベース/スキーマ/テーブルのみ対象にすることは不可能

- プライマリ/スタンバイは同一環境が必須
  - PostgreSQLメジャーバージョン・OSが一致している必要がある
  - 異なるCPUアーキテクチャ間でのレプリケーションも保証されていない？ [1]

[1]: https://www.postgresql.org/message-id/flat/CAAP012m48JhfO8J%3D7ceijFZFKH35oizZPNg3KBhMB-4eTNuE4w%40mail.gmail.com

- 書き込みは常にプライマリのみ
  - スタンバイで実行できるのはトランザクションIDを割り当てず、かつWALを生成しないSQL文だけ

```
standby=# CREATE TABLE foo (id int);
ERROR:  cannot execute CREATE TABLE in a read-only transaction
```

  - SELECT … FOR UPDATE構文や、ROW EXCLUSIVEより強いロックを指定したLOCK構文も不可能

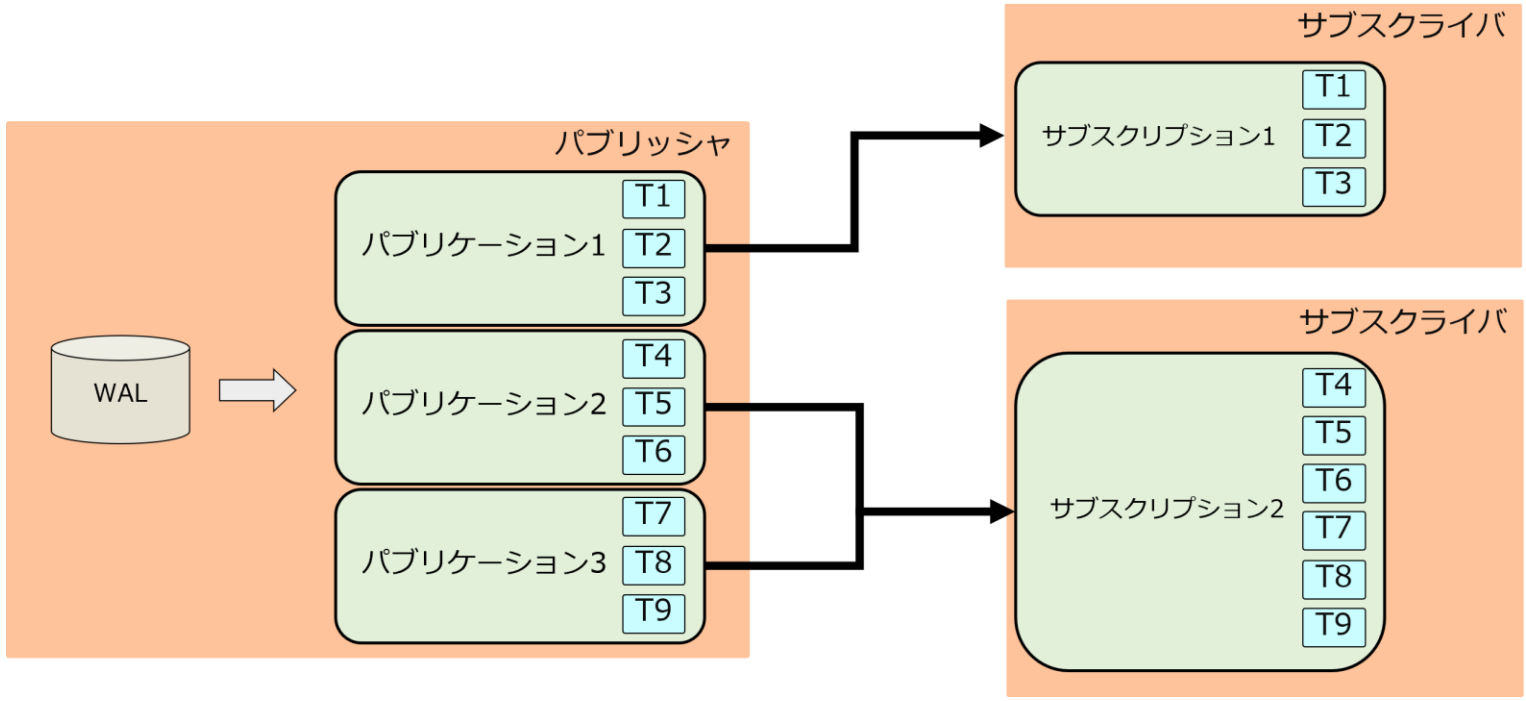
FUJITSU

論理レプリケーション

基本用語
構成・動作
設定パラメータ
環境構築手順
利点と注意点

論理レプリケーションとは？

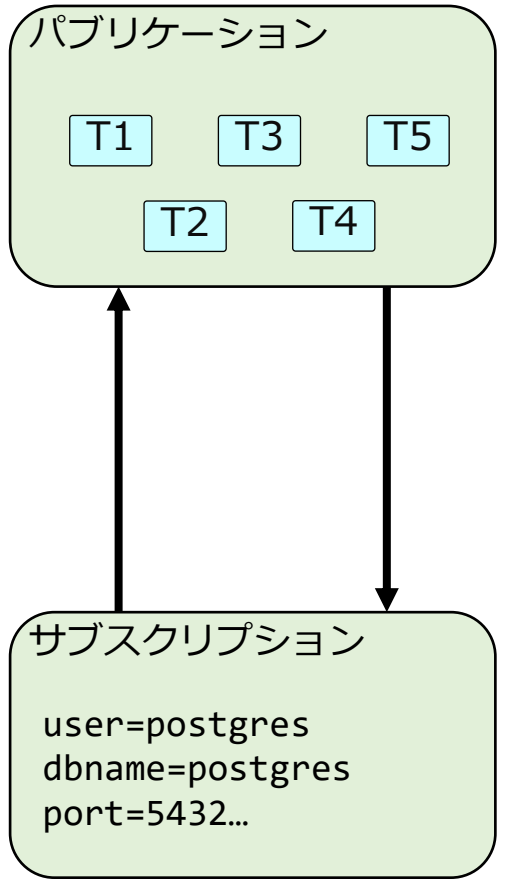
- WALを解析(デコード)し、テーブル単位の変更情報として転送
  - ストリーミングレプリケーションはWALそのまま、論理レプリケーションは関係する内容だけを送る
- 初期のデータ同期と更新追従の2つのフェーズ
- テーブル内の各行をレプリカアイデンティティに基づいて識別する



基本用語

- パブリケーション Publication
  - データと更新を送るレプリケーションの上流
  - 論理レプリケーションの対象となるテーブルと複製するDMLの集合
  - パブリケーションを持つノードをパブリッシャと呼ぶ

- サブスクリプション Subscription
  - レプリケーションの下流
  - パブリッシャへの接続文字列と購読するパブリケーションの集合
  - サブスクリプションを持つノードをサブスクライバと呼ぶ



基本用語

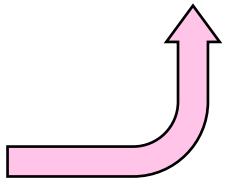
FUJITSU

- レプリカアイデンティティ Replica Identity
  - 2つのノード間で行の一致性を判断するためのオブジェクト
  - UPDATE文やDELETE文を複製するために必要
  - 通常は主キーで、その他ユニークキーや行全体(FULL)を指定することも可能

- 例：給料テーブルの更新

```
publisher=# ¥d employee
              Table "public.employee"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 salary | integer |           |          |
Indexes:
    "employee_pkey" PRIMARY KEY, btree (id)
Publications:
    "pub"
publisher=# UPDATE employee SET salary = 10 WHERE id = 1;
UPDATE 1
```

```
subscriber=# SELECT * FROM employee ;
 id | salary
----+--------
  1 |     10
(1 row)
```



論理レプリケーション環境の構築

FUJITSU

1. パブリケーションの作成

```
publisher=# -- 全テーブルが対象
publisher=# CREATE PUBLICATION all_pub FOR ALL TABLES;
CREATE PUBLICATION

publisher=# -- 特定スキーマ内のテーブルが対象
publisher=# CREATE PUBLICATION schema_pub FOR TABLES IN SCHEMA public ;
CREATE PUBLICATION

publisher=# -- 特定テーブルの条件を満たす行が対象
publisher=# CREATE PUBLICATION filter_pub FOR TABLE employee WHERE (salary > 100);
CREATE PUBLICATION

publisher=# --特定テーブルの特定列が対象
publisher=# CREATE PUBLICATION list_pub FOR TABLE employee (salary);
CREATE PUBLICATION
```

論理レプリケーション環境の構築

FUJITSU

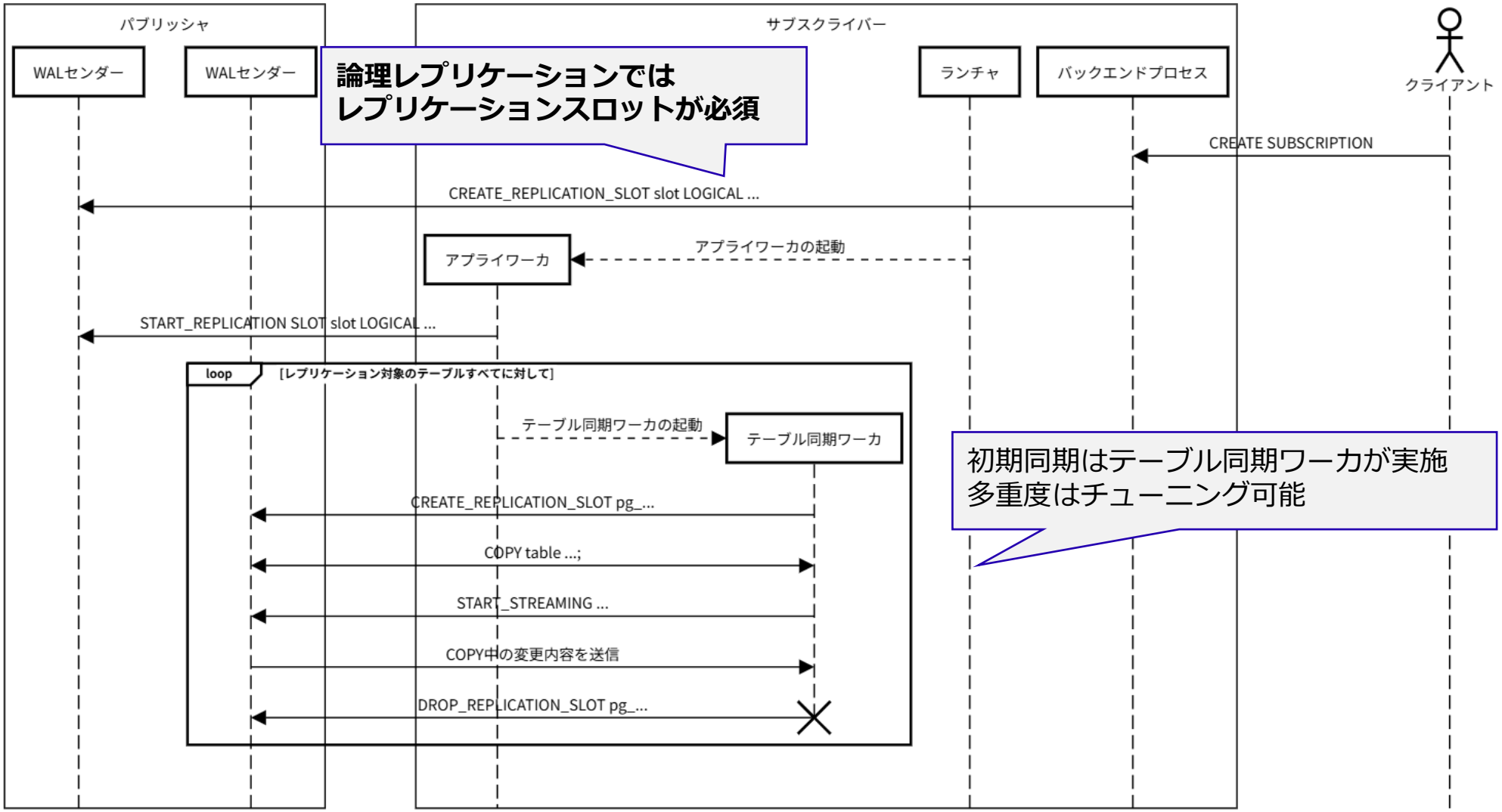
2. サブスクリプションの作成

```
subscriber=# CREATE SUBSCRIPTION all_sub CONNECTION 'user=postgres' PUBLICATION all_pub;
NOTICE:  created replication slot "all_sub" on publisher
CREATE SUBSCRIPTION

subscriber=# -- 複数のパブリケーションを購読
subscriber=# CREATE SUBSCRIPTION sub2 CONNECTION 'user=postgres' PUBLICATION schema_pub, filter_pub;
NOTICE:  created replication slot "sub2" on publisher
CREATE SUBSCRIPTION

subscriber=# -- サブスクリプションパラメータの指定
subscriber=# CREATE SUBSCRIPTION sub3 CONNECTION 'user=postgres' PUBLICATION filter_pub
WITH (binary, two_phase);
NOTICE:  created replication slot "sub3" on publisher
CREATE SUBSCRIPTION
```

レプリケーション接続の確立と初期データ同期



主なパラメータ：論理レプリケーション編

FUJITSU

パラメータ名	解説	設定するインスタンス
wal_level	WAL出力の粒度 **logicalが必須**	パブリッシャー
max_wal_senders	WALセンダープロセスの最大数 サブスクリプションの数 + テーブル同期の並列数に設定	パブリッシャー
max_replication_slots	レプリケーションスロットの最大数 max_wal_sendersと同じかそれ以上に設定	パブリッシャー
max_logical_replication_workers (PG18+) max_replication_slots (PG17-)	レプリケーション起点の数 インスタンスに作成するサブスクリプションの数以上に設定	サブスクライバー
max_logical_replication_workers	必要なアプライワーカ、テーブル同期ワーカ、 パラレルアプライワーカの合計値	サブスクライバー
max_worker_processes	存在できるバックグラウンドワーカプロセスの最大数 max_logical_replication_workers + 1以上	サブスクライバー
max_sync_workers_per_subscription	テーブル同期の並列度	サブスクライバー
max_parallel_apply_workers_per_subscription	大規模トランザクションを並列適用するときの並列度	サブスクライバー

特徴と利点

FUJITSU

- 柔軟な構成が組める
  - 必要なスキーマ・テーブルを選択して複製する
  - 一部SQL(例：INSERT)のみを複製することも可能

```
publisher=# CREATE PUBLICATION pub FOR TABLES IN SCHEMA sch1, sch2 WITH (publish='insert');
CREATE PUBLICATION
```

- 異種環境へのレプリケーションができる
  - 異なるPostgreSQLメジャーバージョンを跨いでデータを共有できる
  - AlmaLinux -> Windows Serverのレプリケーションも可能

- サブスクライバもデータを書き込める
  - 書き込みをスケールさせることができる
  - 双方向レプリケーションを構築することも可能

```
subscriber=# CREATE SUBSCRIPTION sub CONNECTION ‘user=postgres’ PUBLICATION pub WITH (origin=none);
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
```

制約と注意点

- DDLは複製されない
  - テーブルの追加は両インスタンスで実施する必要がある

- データ競合が発生するとレプリケーションは停止する
  - 例：サブスクライバでINSERT文を実行した後、パブリッシャで同一主キー値の行をINSERT
  - 自動的な競合解決機能は存在しない

```
ERROR:  conflict detected on relation "public.foo": conflict=insert_exists
DETAIL:  Key already exists in unique index "foo_pkey", modified in transaction 773.
        Key (id)=(2); existing local row (2); remote row (2).
…
```

- 複製できるのはテーブルのみ
  - シーケンス・ラージオブジェクト・マテリアライズドビューはレプリケーションの対象外

論理レプリケーションのアップデート

FUJITSU

≤10 (2017)
- 論理レプリケーションのサポート
- レプリカアイデンティティの導入
- レプリケーションスロットの導入

11 (2018)
- TRUNCATE文のレプリケーション

12 (2019)
- レプリケーションスロットのコピー

13 (2020)
- logical_decoding_work_memの導入
- max_slot_wal_keep_sizeの導入
- パーティションテーブルのレプリケーションが可能に

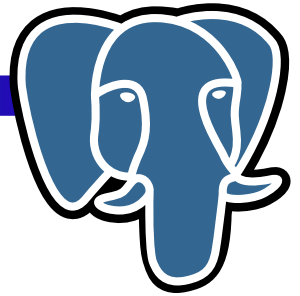
14 (2021)
- 大規模トランザクションのストリーム
- バイナリ転送モードの導入

15 (2022)
- 2PCがレプリケーションが可能に
- FOR ALL TABLES IN SCHEMA パブリケーションの導入
- 行フィルタ/列リストの導入
- エラー時にサブスクリプションを無効化

16 (2023)
- スタンバイからの論理レプリケーションが可能に
- 双方向レプリケーションが可能に
- 大規模トランザクションの並列適用が可能に

17 (2024)
- レプリケーションスロットの同期
- pg_upgrade対応
- pg_createsubscriberの導入

18 (2025)
- 競合検知
- 生成列のレプリケーション
- pg_createsubscriber改善



レプリケーション比較

比較表
ストリーミングレプリケーションのユースケース
論理レプリケーションのユースケース

レプリケーションの比較

比較項目	ストリーミングレプリケーション	論理レプリケーション
上流/下流インスタンスの呼び方	プライマリ/スタンバイ	パブリッシャ/サブスクライバ
下流に送る内容	**WALそのもの**	**WALから抽出された変更情報**
複製する対象	インスタンス全体	テーブル・スキーマ単位で指定 最大でもDB全体
上流インスタンスの数	1つだけ	複数設定が可能 サブスクリプションを複数持つことも可能
インスタンスの環境差異	同一OS・メジャーバージョンのみ	異なるOS・メジャーバージョン間でも可
制限事項	スタンバイでデータ更新不可	一部データ(DDL・シーケンス etc)は複製不可

プライマリ/スタンバイで完全に同一
HA構成による可用性向上

異種環境でも動作
アップグレード時のデータ移行

ストリーミングレプリケーションの例：HA構成

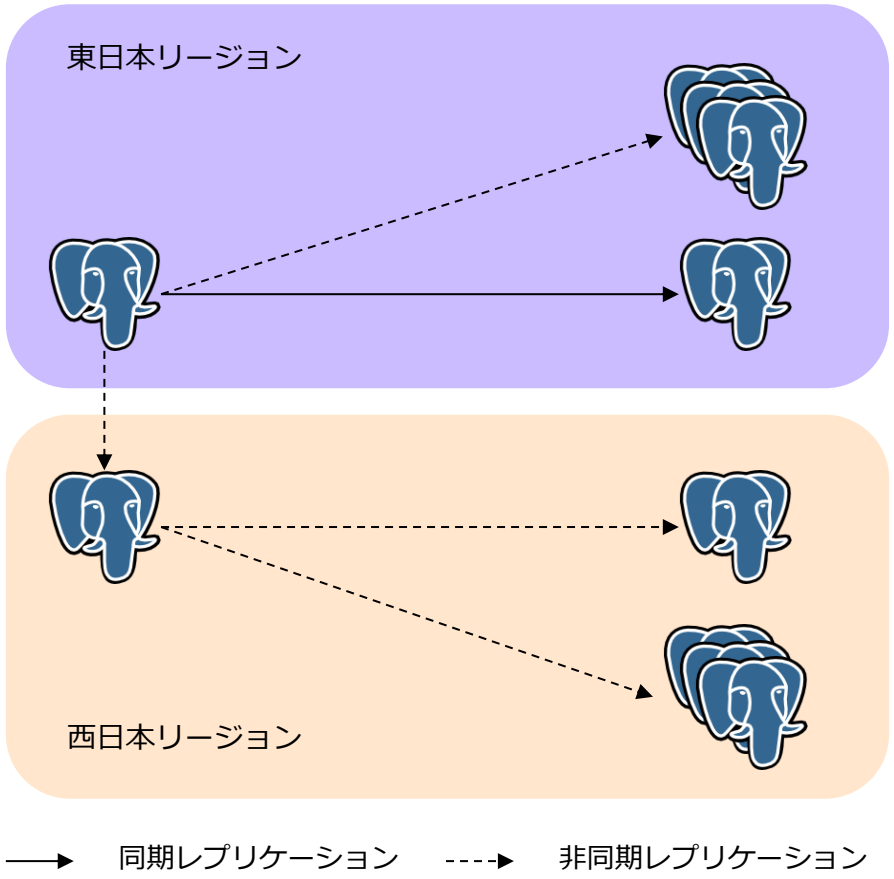
FUJITSU

- 同一のデータを持つインスタンスを複数リージョンに作成して、可用性を高める手法

- 運用リージョン内には1つ以上の同期レプリカを設定
  - プライマリの故障時は基本的にこのマシンが次のプライマリになる
  - 非同期レプリケーションでリードレプリカも作成可能

- リージョン間のレプリケーションは極力減らす
  - 地理的に遠いのでレプリケーションが遅延しやすい
  - リージョン間レプリケーションは1つだけで、それ以外はリージョン内で非同期レプリカに設定するとよい

- レプリケーション進捗が同期 < 非同期となることがある
  - プライマリ故障時に別リージョンにだけWALが届いているケース
  - こうなると新プライマリ->別リージョン間のレプリケーションが構築できない
  - 別リージョンのインスタンスを再構築するしかない



論理レプリケーションの例：アップグレード

FUJITSU

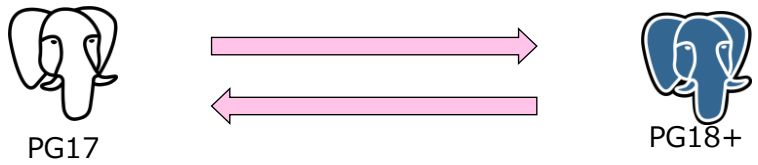
- 業務を継続したまま、DBインスタンス内のデータを新環境に移せる
  - 旧インスタンスにパブリケーションを、新インスタンスにサブスクリプションを作成

```
old=# CREATE PUBLICATION pub FOR ALL TABLES;
CREATE PUBLICATION
```

```
new=# CREATE SUBSCRIPTION sub1 CONNECTION 'user=postgres port=5432' PUBLICATION pub ;
NOTICE:  created replication slot "sub1" on publisher
CREATE SUBSCRIPTION
```

  - データが自動的に同期され、自動的にデータ移行が完了する

- 同期中に行われたデータの追加や変更も自動的に反映される



- 切り戻しに対応することも可能
  - データ同期が完了した後、旧系->新系の論理レプリケーションを構築
  - 新インスタンスへの変更を旧インスタンスにも反映させる
  - 業務開始後に問題が発生した場合は旧インスタンスに再接続

その他

ロールと認証

監視

ロールと認証

- レプリケーション接続に使用するロールにはREPLICATION属性とLOGIN属性が必要
  - ストリーミング/論理レプリケーションの両方
  - この権限がないとプライマリ/パブリッシャに接続ができない

```
$ pg_basebackup -d "user=test" -D standby -R
pg_basebackup: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed:
FATAL:  permission denied to start WAL sender
DETAIL:  Only roles with the REPLICATION attribute may start a WAL sender process.
```

- pg_hba.conf
  - ストリーミングレプリケーション：database列に**replication**と書かれたレコードが合致する
  - 論理：通常と同じ。database列に**複製対象のデータベース名**が書かれたレコードが合致する

論理レプリケーションと権限

操作	必要な権限
パブリケーションの作成	データベースへのCREATE権限
サブスクリプションの作成	データベースへのCREATE権限とpg_create_subscriptionロール
テーブルをパブリケーションに追加する	テーブルの所有権
データベース/スキーマ内の全テーブルをパブリケーションに追加する	**スーパーユーザ**

- パブリッシャは行セキュリティポリシーの影響を受ける
  - 回避するにはレプリケーション接続に使用したロールにSUPERUSER属性とBYPASSRLS属性が必要

- アプライワーカは**テーブル所有者のロール**を用いて変更を適用する
  - run_as_ownerオプションを付けるとサブスクリプション所有者の権限で変更を適用するようになる
    適切な権限を持っていないと変更適用に失敗する

```
subscriber=# CREATE SUBSCRIPTION sub CONNECTION 'user=postgres' PUBLICATION pub WITH (run_as_owner=true);
```

監視するべきビューとシステムカタログ

FUJITSU

パラメータ名	解説	確認するインスタンス
pg_stat_replication	WALの送信に関する統計情報	プライマリ/パブリッシャ
pg_stat_wal_receiver	WALの受信に関する統計情報	スタンバイ
pg_stat_wal	wal_buffers_full列はWALバッファが満杯になった回数を示す 多い場合はwal_buffersを調整して最適なバッファサイズに変更	プライマリ/パブリッシャ
pg_stat_database_conflicts	conf_x列は競合によるクエリキャンセルが発生した回数を示す max_standby_streaming_delayを長く設定すると回避可能？	スタンバイ
pg_stat_replication_slots	spill_x列はデコード中にメモリが満杯になった回数を示す 多い場合はlogical_decoding_work_memを増やすか、 大規模トランザクションのストリームを有効にする	プライマリ/パブリッシャ
pg_stat_subscription_stats	適用ワーカの変更適用中に起こったエラー回数を記録 ゼロじゃない場合はアプリケーションの見直しが必要	サブスクライバ
pg_stat_subscription	変更を受信した時刻を記録 遅れている場合はネットワーク	サブスクライバ
pg_subscrltion_rel	同期が終了したらsrsubstate列の値はrになる ならない場合はデータ競合が発生している可能性	サブスクライバ

監視方法の例：ストリーミングレプリケーション

FUJITSU

- pg_stat_replicationのsent_lsnとpg_current_wal_lsn()の差が大きい
  - ……プライマリサーバが高負荷で生成されたWALを転送しきれていない
  - wal_compressionをonに設定して、WALの分量を減らす？

```
primary=# SELECT sent_lsn, write_lsn, flush_lsn, replay_lsn, pg_current_wal_lsn() FROM pg_stat_replication;
  sent_lsn   | write_lsn  | flush_lsn  | replay_lsn | pg_current_wal_lsn
-------------+------------+------------+------------+--------------------
 0/030A2000 | 0/0304D958 | 0/0304D958 | 0/0304D958 | 0/06000000
(1 row)
```

- pg_stat_wal_receiverのflushed_lsnとpg_last_wal_replay_lsn()の差が大きい
  - ……スタンバイサーバが高負荷で転送されたWALを反映しきれていない

```
standby=# SELECT flushed_lsn, pg_last_wal_replay_lsn() FROM pg_stat_wal_receiver;
 flushed_lsn | pg_last_wal_replay_lsn
-------------+-----------------------
 0/06000060  | 0/030480E8
(1 row)
```

監視方法の例：論理レプリケーション

FUJITSU

- pg_stat_replication_slotsのrestart_lsnやconfirmed_flush_lsnとpg_current_wal_lsn()の差が大きい
  - サブスクライバが変更に追い付いていない
  - このままだとWALが残り続け、diskを圧迫する可能性あり

```
publisher=# SELECT restart_lsn, confirmed_flush_lsn, pg_current_wal_lsn() FROM pg_replication_slots ;
 restart_lsn | confirmed_flush_lsn | pg_current_wal_lsn
-------------+---------------------+--------------------
 0/030000D8  | 0/03000110          | 0/06000060
(1 row)
```

  - サブスクライバのpg_stat_subscription_statsを確認し、データ競合が発生しているかどうか確認する
  - 起きていた場合は手動で解決するか、ALTER SUBSCRIPTION SKIP構文を使用してトランザクションの適用をスキップ

```
subscriber=# SELECT subname, apply_error_count, confl_insert_exists FROM pg_stat_subscription_stats;
 subname | apply_error_count | confl_insert_exists
---------+-------------------+---------------------
 sub     |                40 |                  40
(1 row)
```

まとめ・参考資料

FUJITSU

- PostgreSQLには2種のレプリケーションが存在
- HA構成にはストリーミングレプリケーション・データ移行には論理レプリケーションがオススメ

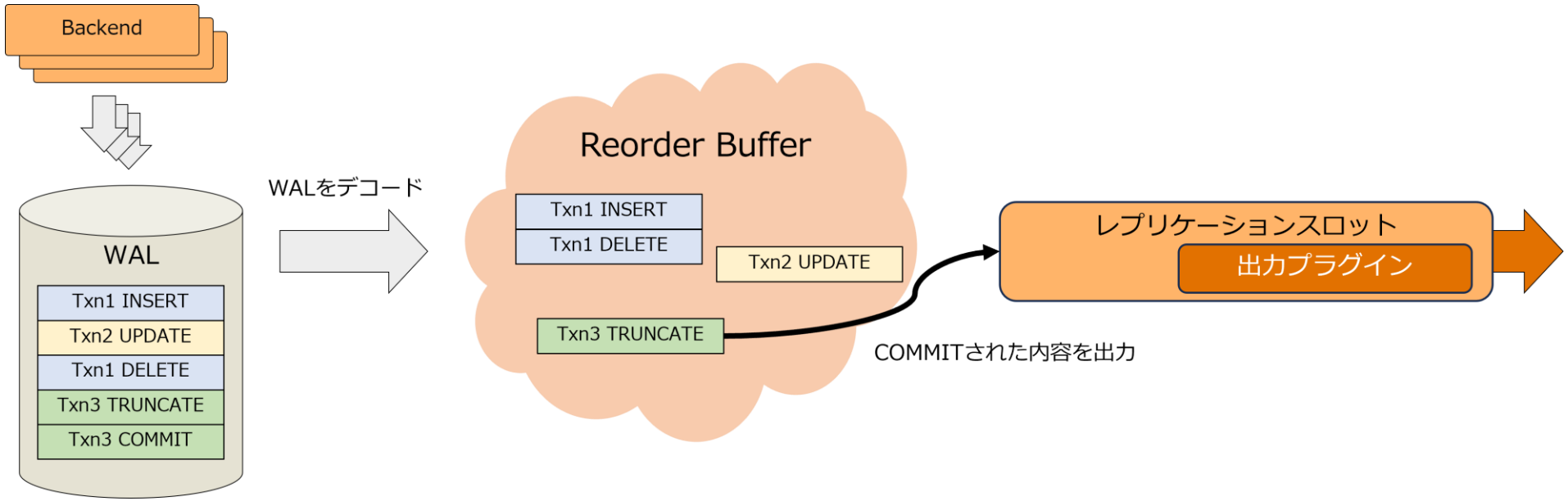
- PostgreSQL 日本語ドキュメント
  - https://www.postgresql.jp/document/current/index.html

- The Internals of PostgreSQL
  - https://www.interdb.jp/pg/

- PostgreSQL Operations cheat sheet
  - https://wiki.postgresql.org/wiki/Operations_cheat_sheet

論理デコーディング

- データベース内の更新内容を外部に出力する仕組み
- WALをデコードして関係する更新だけを抽出する
- 出力フォーマットはプラグインとして設定可能
  - 公式リポジトリには2種のサンプルプラグインが同梱
  - APIは公開されているので自作も可能



論理デコーディング | 論理レプリケーションスロット

FUJITSU

- どこまでのWALを読み取って出力したのかを記録
  - Checkpoint時に未デコードのWALファイルが削除されるのを防ぐ
- スロット名と出力プラグイン名を指定して作成

不要なスロットを放置していると、WALファイルがリサイクル削除されずにディスクが一杯になるかも……

```
postgres=# SELECT * FROM pg_create_logical_replication_slot('test', 'test_decoding');
 slot_name |    lsn
-----------+-----------
 test      | 0/1506180
(1 row)
postgres=# SELECT slot_name, plugin, database, confirmed_flush_lsn
             FROM pg_replication_slots;
 slot_name |    plugin     | database | confirmed_flush_lsn
-----------+---------------+----------+---------------------
 test      | test_decoding | postgres | 0/1506180
(1 row)
```

論理デコーディング | 更新内容の出力

FUJITSU

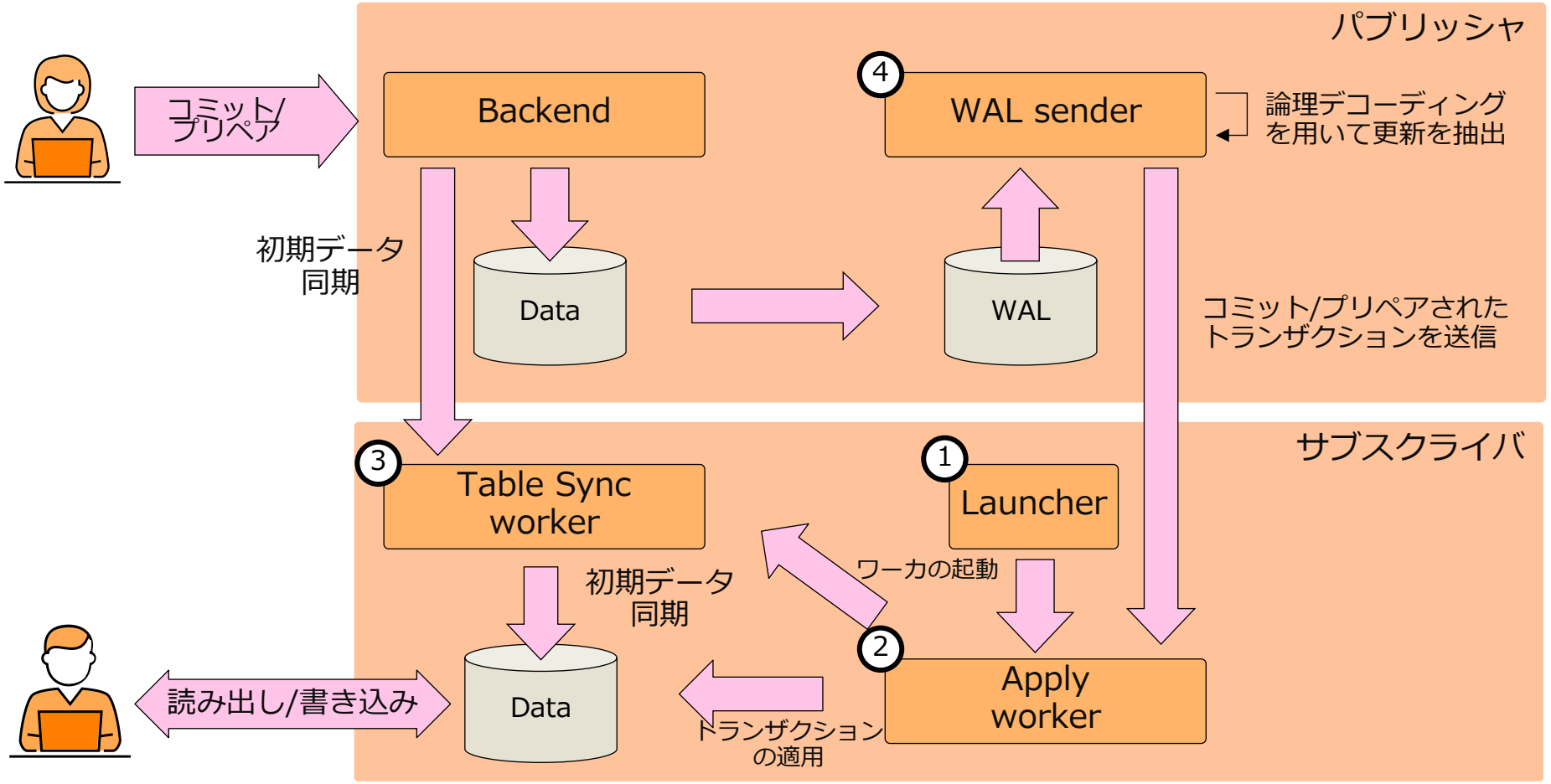
- トランザクションがコミットされると、更新内容を抽出できる

```
postgres=# INSERT INTO tbl VALUES (generate_series(1, 2));
INSERT 0 2
postgres=# SELECT * FROM pg_logical_slot_get_changes('test', NULL, NULL);
    lsn    | xid |                   data
-----------+-----+-----------------------------------------
 0/15061B8 | 734 | BEGIN 734
 0/15061B8 | 734 | table public.tbl: INSERT: id[integer]:1
 0/1506298 | 734 | table public.tbl: INSERT: id[integer]:2
 0/1506348 | 734 | COMMIT 734
(4 rows)
```

- DDLを抽出することはできない

```
postgres=# CREATE TABLE foo (id int);
CREATE TABLE
postgres=# SELECT * FROM pg_logical_slot_get_changes('test', NULL, NULL);
    lsn    | xid |    data
-----------+-----+------------
 0/15063B0 | 735 | BEGIN 735
 0/150D950 | 735 | COMMIT 735
(2 rows)
```

プロセスアーキテクチャ

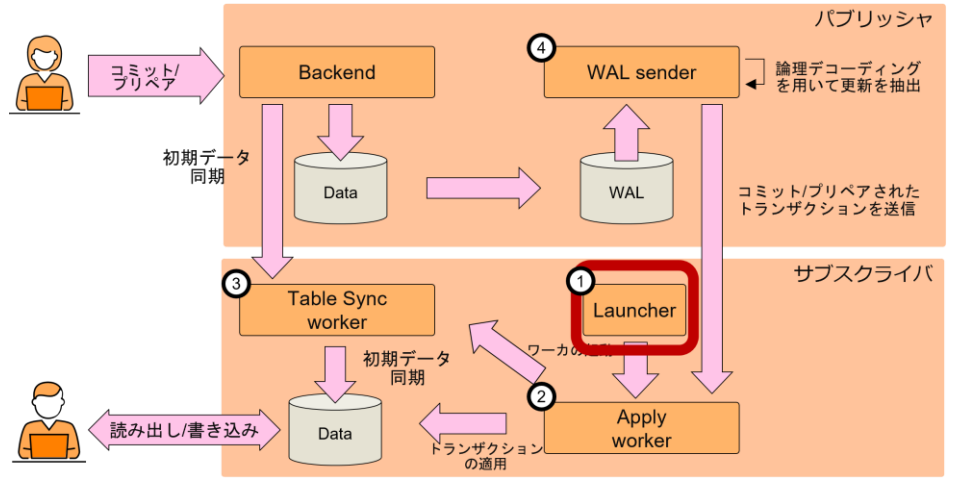


Launcher

FUJITSU

- サーバプロセス起動と同時に起動するバックグラウンドワーカ
- 定期的にpg_subscriptionを確認し、各行に対応するapply workerを生成

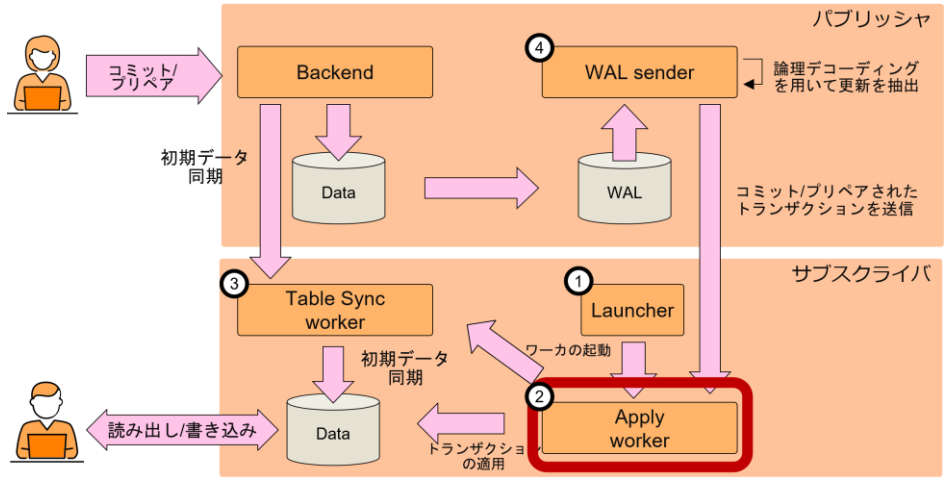
```
postgres=# SELECT datid, pid, backend_type FROM pg_stat_activity
             WHERE backend_type LIKE 'logical%';
 datid |  pid  |         backend_type
-------+-------+------------------------------
       | 22427 | logical replication launcher
(1 row)
```



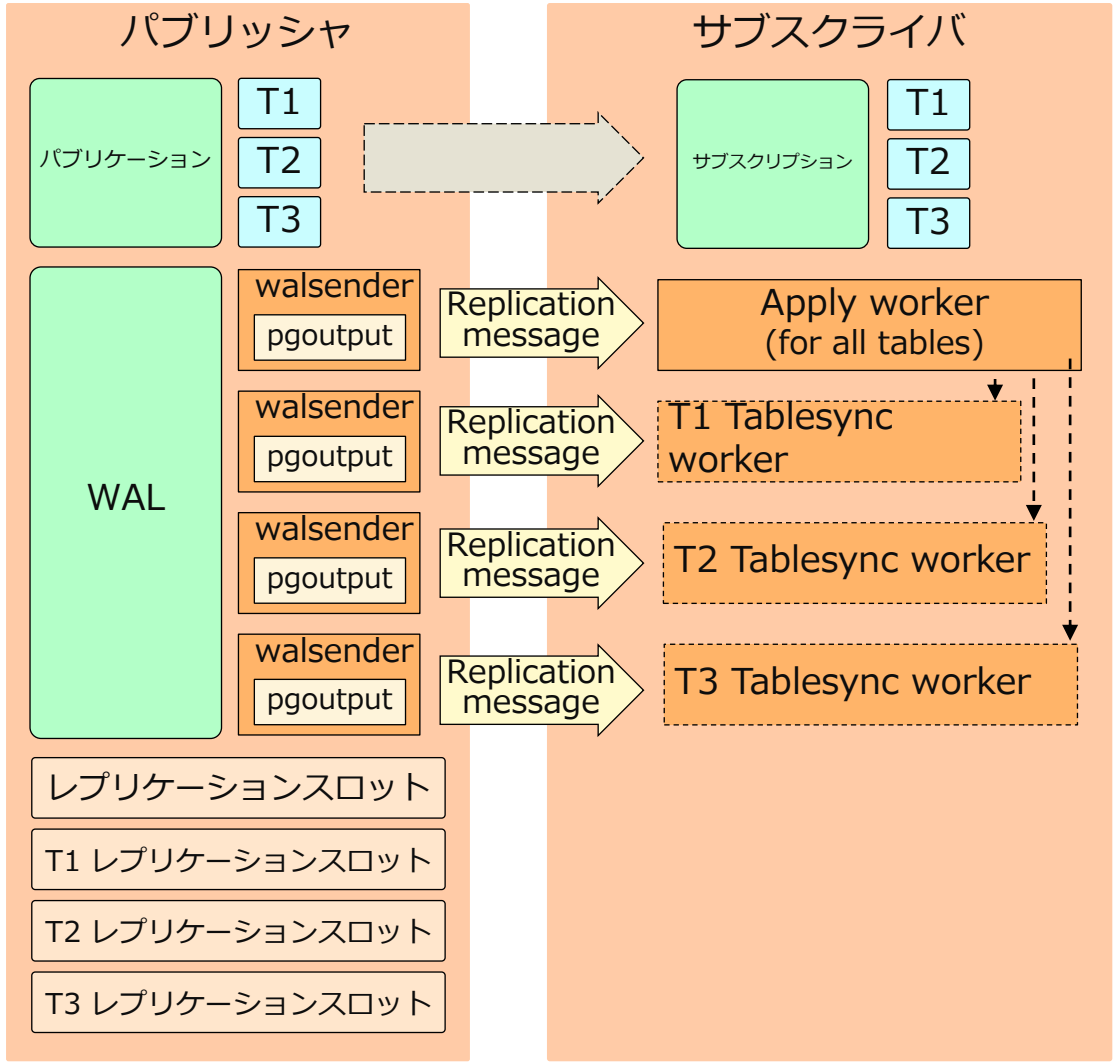
Apply worker

- サブスクライバ側にのみ存在するバックグラウンドワーカ
- pg_subscriptionの毎行に1つ存在
- パブリッシャに接続し、WAL senderからの出力内容を受け取って適用する
  - パブリッシャ上に論理レプリケーションスロットを作成し、そこから論理デコーディングを用いて更新内容を抽出する
- pg_stat_subscriptionビューでワーカの状態を参照可能

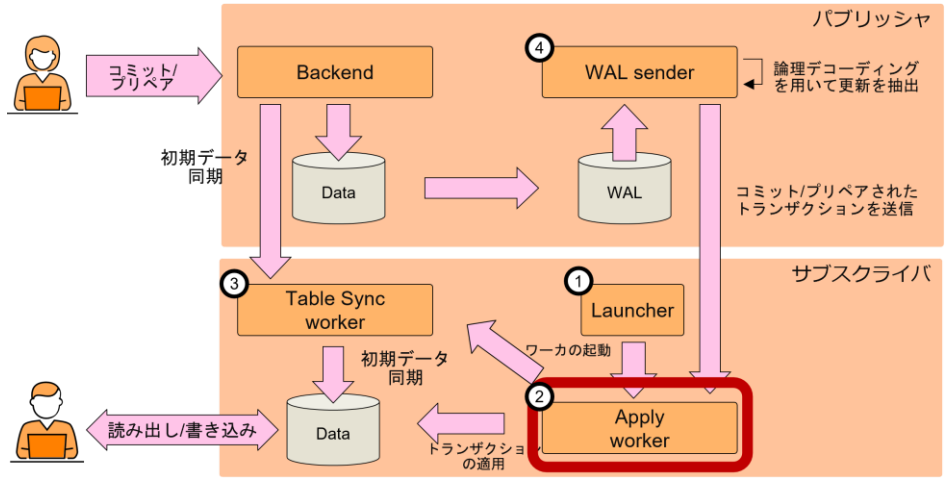
```
postgres=# SELECT subname, received_lsn
             FROM pg_stat_subscription;
 subname | received_lsn
---------+--------------
 sub     | 0/1546788
(1 row)
```



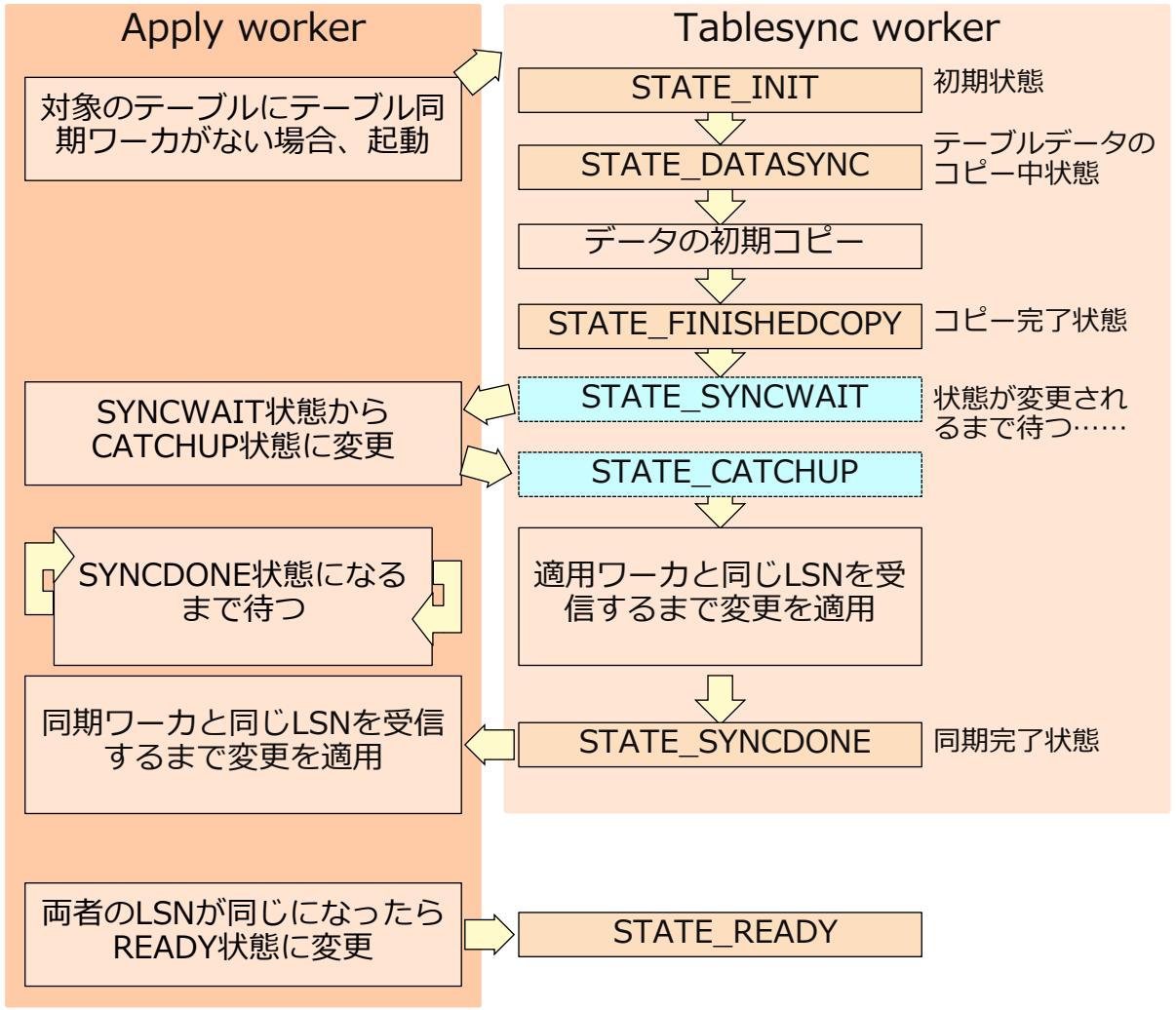
Apply worker



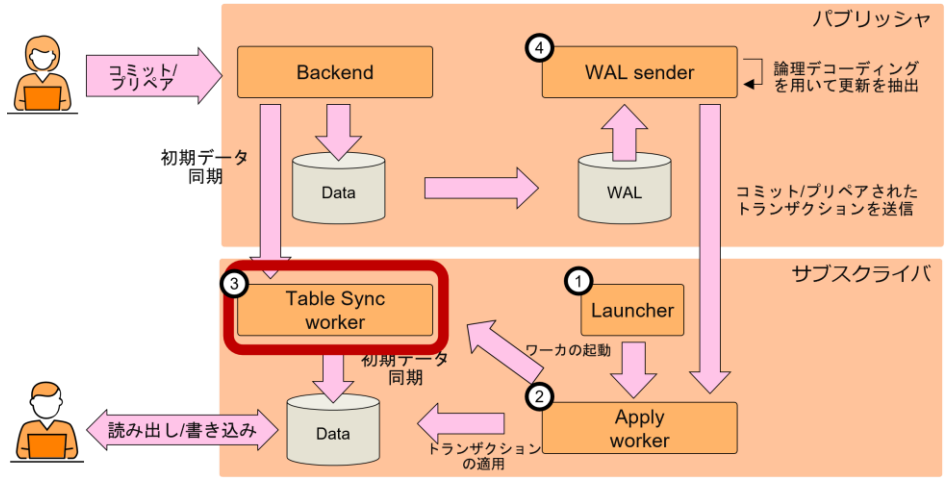
- 初期同期時には、対象となったテーブル毎にTablesync workerを起動する
- それぞれのTablesync workerがレプリケーションスロットを作成し、対応するWAL senderから更新を受け取って適用する



Tablesync worker



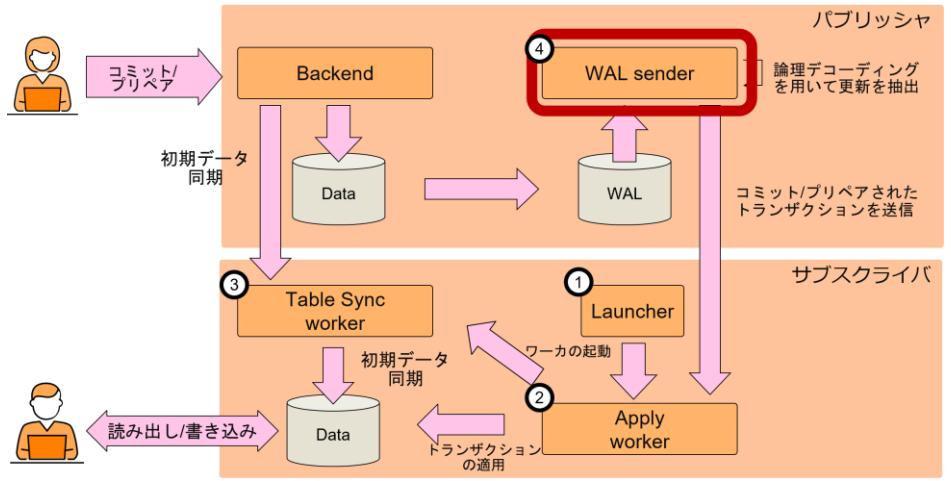
- サブスクライバ側にのみ存在するバックグラウンドワーカ
- テーブル毎に生成される
- COPY文による初期コピーと、ストリーミングプロトコルによる更新の受信・適用の2つのフェーズからなる



WAL sender

- パブリッシャ側にのみ存在するバックエンドプロセス
- 適用ワーカ/同期ワーカの要求に応じて起動
- 論理デコーディングの仕組みを用いてWALをデコードし、出力内容をワーカに送信し続ける
- pg_stat_replication_slotsビューでスロットの使用状況を参照可能

```
postgres=# SELECT slot_name, total_txns, total_bytes
FROM pg_stat_replication_slots;
 slot_name | total_txns | total_bytes
-----------+------------+-------------
 sub       |         56 |        7392
(1 row)
```



論理レプリケーション環境の構築

FUJITSU

1. パブリッシャ・サブスクライバのパラメータ設定
2. パブリケーションの作成

```
publisher=# CREATE TABLE foo (id INT PRIMARY KEY);
CREATE TABLE
publisher=# INSERT INTO foo VALUES (generate_series(1, 5));
INSERT 0 5
publisher=# CREATE PUBLICATION pub FOR TABLE foo ;
CREATE PUBLICATION
```

3. サブスクリプションの作成

```
subscriber=# CREATE TABLE foo (id int PRIMARY KEY);
CREATE TABLE
subscriber=# CREATE SUBSCRIPTION sub CONNECTION 'port=5432 user=postgres' PUBLICATION pub;
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
subscriber =# SELECT count(*) FROM foo ;
 count
-------
     5
(1 row)
```

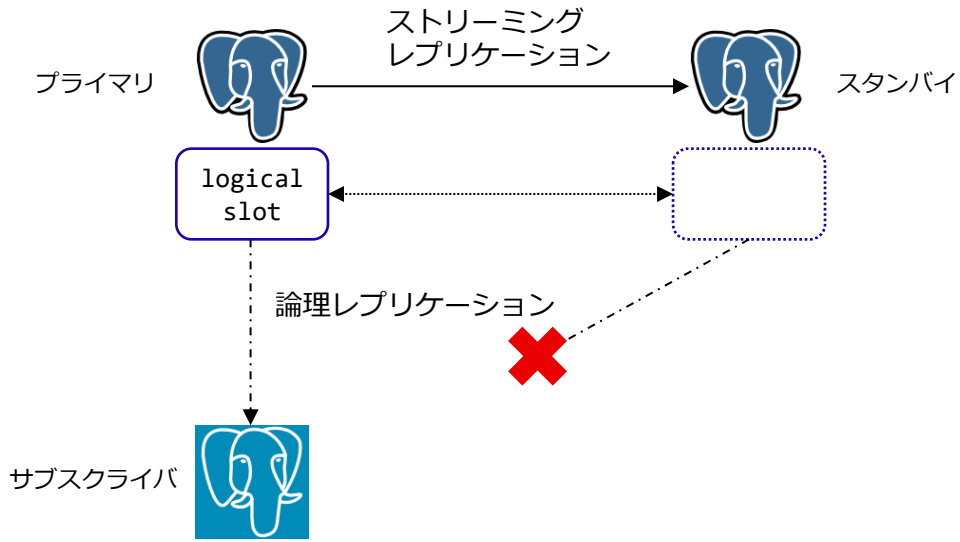
論理レプリケーションスロットの同期

FUJITSU

- パブリッシャを複製する場合、スタンバイにも論理スロットが必要
  - サブスクライバが昇格インスタンスに接続して、論理レプリケーションを再開できるようにしたい
  - このためには対応する論理スロットが昇格インスタンスにお必要

- ……しかし、スロットそのものはレプリケーションで複製できなかった
  - 手動でスロットを作成することはできるが、状態が更新されないのでWALが貯まり続ける

スロット同期機能が導入
プライマリ-スタンバイ間で必要なWALの情報が同期



スロット同期の対象

FUJITSU

○failoverオプションを指定した論理スロットが同期の対象になる

```
subscriber=# CREATE SUBSCRIPTION sub CONNECTION 'user=postgres' PUBLICATION pub WITH (failover=true);
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
```

```
publisher=# SELECT pg_create_logical_replication_slot('test', 'test_decoding', false, false, true);
 pg_create_logical_replication_slot
------------------------------------
 (test,0/018A9210)
(1 row)

publisher=# SELECT slot_name, plugin, failover FROM pg_replication_slots ;
 slot_name |    plugin     | failover
-----------+---------------+----------
 sub       | pgoutput      | t
 test      | test_decoding | t
(2 rows)
```

スロット同期の方法

FUJITSU

- SQL関数pg_sync_replication_slotsを実行
  - 関数実行のタイミングで同期的に動作

```
postgres=# SELECT pg_sync_replication_slots();
 pg_sync_replication_slots
---------------------------

(1 row)
postgres=# SELECT slot_name, failover, synced FROM pg_replication_slots;
 slot_name | failover | synced
-----------+----------+--------
 test      | t        | t
(1 row)
```

- sync_replication_slotsをonに設定
  - スロット同期ワーカが非同期的・周期的に動作

```
$ ps auxf | grep postgres
…
|       ¥_ /usr/local/pgsql/bin/postgres -D standby
…
|           ¥_ postgres: slotsync worker
```

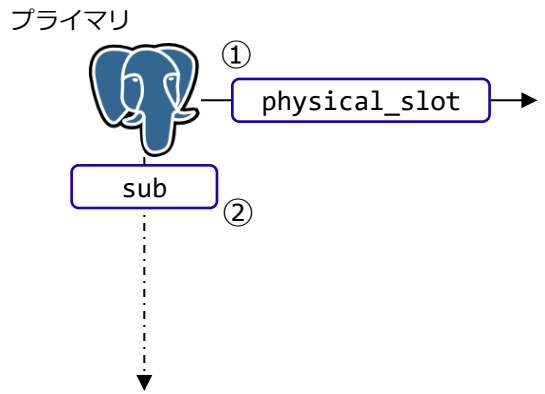
スロット同期の注意

- スタンバイのwal_levelをlogicalに設定する
  - 論理スロットの作成に必須

- ストリーミングレプリケーションに物理レプリケーションスロット使用する
  - スタンバイ->サブスクライバの順でデータが複製されることを保証するのに使う
  - どうやって？ - プライマリ->スタンバイ間の物理スロットを**synchronized_standby_slots**に設定

```
primary=# SELECT slot_name, slot_type, failover from pg_replication_slots ;
   slot_name    | slot_type | failover
----------------+-----------+----------
 physical_slot | physical  | f
 sub            | logical   | t
(2 rows)

primary=# SHOW synchronized_standby_slots ;
 synchronized_standby_slots
----------------------------
 physical_slot
(1 row)
```



論理レプリケーションと権限

FUJITSU

操作	必要な権限
パブリケーションの作成	データベースへのCREATE権限
サブスクリプションの作成	データベースへのCREATE権限とpg_create_subscriptionロール
テーブルをパブリケーションに追加する	テーブルの所有権
データベース/スキーマ内の全テーブルをパブリケーションに追加する	**スーパーユーザ**

- パブリッシャは行セキュリティポリシーの影響を受ける
  - 回避するにはレプリケーション接続に使用したロールにSUPERUSER属性とBYPASSRLS属性が必要

- アプライワーカは**テーブル所有者のロール**を用いて変更を適用する
  - run_as_ownerオプションを付けるとサブスクリプション所有者の権限で変更を適用するようになる
    適切な権限を持っていないと変更適用に失敗する

```
subscriber=# CREATE SUBSCRIPTION sub CONNECTION 'user=postgres' PUBLICATION pub WITH (run_as_owner=true);
```